# Lenora College of Engineering

# Rampachodavaram

# M. Tech R19 C.S.E

# 1st Year 1st Semester

# Advanced Data Structures and Algorithms Lab

**Program 1**

**Java Program to create and display a singly linked list**

Algorithm

- Create a class Node which has two attributes: data and next. Next is a pointer to the next node.
- Create another class which has two attributes: head and tail.
- addNode() will add a new node to the list:
    - Create a new node.
    - It first checks, whether the head is equal to null which means the list is empty.
    - If the list is empty, both head and tail will point to the newly added node.
    - If the list is not empty, the new node will be added to end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.

a. display() will display the nodes present in the list:

- Define a node current which initially points to the head of the list.
- Traverse through the list till current points to null.
- Display each node by making current to point to node next to it in each iteration.

---

Program:

```
1. public class SinglyLinkedList {
2.    //Represent a node of the singly linked list
3.    class Node{
4.       int data;
5.       Node next;
6.
```

```
7.      public Node(int data) {
8.          this.data = data;
9.          this.next = null;
10.     }
11. }
12.
13. //Represent the head and tail of the singly linked list
14. public Node head = null;
15. public Node tail = null;
16.
17. //addNode() will add a new node to the list
18. public void addNode(int data) {
19.     //Create a new node
20.     Node newNode = new Node(data);
21.
22.     //Checks if the list is empty
23.     if(head == null) {
24.         //If list is empty, both head and tail will point to new node
25.         head = newNode;
26.         tail = newNode;
27.     }
28.     else {
29.         //newNode will be added after tail such that tail's next will point to newNode
30.         tail.next = newNode;
31.         //newNode will become new tail of the list
32.         tail = newNode;
33.     }
34. }
35.
36. //display() will display all the nodes present in the list
37. public void display() {
38.     //Node current will point to head
39.     Node current = head;
40.
41.     if(head == null) {
42.         System.out.println("List is empty");
43.         return;
44.     }
45.     System.out.println("Nodes of singly linked list: ");
46.     while(current != null) {
47.         //Prints each node by incrementing pointer
48.         System.out.print(current.data + " ");
49.         current = current.next;
50.     }
51.     System.out.println();
52. }
53.
```

```
54.    public static void main(String[] args) {
55.
56.        SinglyLinkedList sList = new SinglyLinkedList();
57.
58.        //Add nodes to the list
59.        sList.addNode(1);
60.        sList.addNode(2);
61.        sList.addNode(3);
62.        sList.addNode(4);
63.
64.        //Displays the nodes present in the list
65.        sList.display();
66.    }
67. }
```

**Output:**

*Nodes of singly linked list:*
*1 2 3 4*

**Program 2**

**a. Java Program for Reverse a linked list**

```
// Java program for reversing the Linked list


class LinkedList {


    static Node head;


    static class Node {


        int data;

        Node next;


        Node(int d) {

            data = d;

            next = null;

        }

    }


    // A simple and tail recursive function to reverse

    // a linked list. prev is passed as NULL initially.

    Node reverseUtil(Node curr, Node prev) {


        /* If last node mark it head*/

        if (curr.next == null) {

            head = curr;
```

```
                /* Update next to prev node */
                curr.next = prev;
                return null;
        }

        /* Save curr->next node for recursive call */
        Node next1 = curr.next;

        /* and update next ..*/
        curr.next = prev;

        reverseUtil(next1, curr);
        return head;
}

// prints content of double linked list
void printList(Node node) {
        while (node != null) {
                System.out.print(node.data + " ");
                node = node.next;
        }
}

public static void main(String[] args) {
        LinkedList list = new LinkedList();
```

```
                list.head = new Node(1);
                list.head.next = new Node(2);
                list.head.next.next = new Node(3);
                list.head.next.next.next = new Node(4);
                list.head.next.next.next.next = new Node(5);
                list.head.next.next.next.next.next = new Node(6);
                list.head.next.next.next.next.next.next = new Node(7);
                list.head.next.next.next.next.next.next.next = new Node(8);

                System.out.println("Original Linked list ");
                list.printList(head);
                Node res = list.reverseUtil(head, null);
                System.out.println("");
                System.out.println("");
                System.out.println("Reversed linked list ");
                list.printList(res);
        }
    }
```

**Output**

```
Original Linked list
1 2 3 4 5 6 7 8

Reversed linked list
8 7 6 5 4 3 2 1
```

**Time Complexity:** O(N), where N is the length of the given linked list
**Auxiliary Space:** O(1)

## b. Program to sort the elements of the singly linked list

Algorithm

1. Create a class Node which has two attributes: data and next. Next is a pointer to the next node in the list.
2. Create another class SortList which has two attributes: head and tail.
3. addNode() will add a new node to the list:
    1. Create a new node.
    2. It first checks, whether the head is equal to null which means the list is empty.
    3. If the list is empty, both head and tail will point to a newly added node.
    4. If the list is not empty, the new node will be added to end of the list such that tail's next will point to a newly added node. This new node will become the new tail of the list.
4. sortList() will sort the nodes of the list in ascending order.
    1. Define a node current which will point to head.
    2. Define another node index which will point to node next to current.
    3. Compare data of current and index node. If current's data is greater than the index's data then, swap the data between them.
    4. Current will point to current.next and index will point to index.next.
    5. Continue this process until the entire list is sorted.
5. display() will display the nodes present in the list:
    1. Define a node current which will initially point to the head of the list.
    2. Traverse through the list till current points to null.
    3. Display each node by making current to point to node next to it in each iteration.

```
1.  public class SortList {
2.
3.      //Represent a node of the singly linked list
4.      class Node{
5.          int data;
6.          Node next;
7.
8.          public Node(int data) {
9.              this.data = data;
10.             this.next = null;
11.         }
12.     }
13.
14.     //Represent the head and tail of the singly linked list
15.     public Node head = null;
```

```
16.    public Node tail = null;
17.
18.    //addNode() will add a new node to the list
19.    public void addNode(int data) {
20.       //Create a new node
21.       Node newNode = new Node(data);
22.
23.       //Checks if the list is empty
24.       if(head == null) {
25.          //If list is empty, both head and tail will point to new node
26.          head = newNode;
27.          tail = newNode;
28.       }
29.       else {
30.          //newNode will be added after tail such that tail's next will point to newNode
31.          tail.next = newNode;
32.          //newNode will become new tail of the list
33.          tail = newNode;
34.       }
35.    }
36.
37.    //sortList() will sort nodes of the list in ascending order
38.    public void sortList() {
39.       //Node current will point to head
40.       Node current = head, index = null;
41.       int temp;
42.
43.       if(head == null) {
44.          return;
45.       }
46.       else {
47.          while(current != null) {
48.             //Node index will point to node next to current
49.             index = current.next;
50.
51.             while(index != null) {
52.                //If current node's data is greater than index's node data, swap the data between
    them
53.                if(current.data > index.data) {
54.                   temp = current.data;
55.                   current.data = index.data;
56.                   index.data = temp;
57.                }
58.                index = index.next;
59.             }
60.             current = current.next;
61.          }
```

```
62.       }
63.    }
64.
65.    //display() will display all the nodes present in the list
66.    public void display() {
67.       //Node current will point to head
68.       Node current = head;
69.       if(head == null) {
70.          System.out.println("List is empty");
71.          return;
72.       }
73.       while(current != null) {
74.          //Prints each node by incrementing pointer
75.          System.out.print(current.data + " ");
76.          current = current.next;
77.       }
78.       System.out.println();
79.    }
80.
81.    public static void main(String[] args) {
82.
83.       SortList sList = new SortList();
84.
85.       //Adds data to the list
86.       sList.addNode(9);
87.       sList.addNode(7);
88.       sList.addNode(2);
89.       sList.addNode(5);
90.       sList.addNode(4);
91.
92.       //Displaying original list
93.       System.out.println("Original list: ");
94.       sList.display();
95.
96.       //Sorting list
97.       sList.sortList();
98.
99.       //Displaying sorted list
100.             System.out.println("Sorted list: ");
101.             sList.display();
102.          }
103.       }
```

**Output:**

*Original list:*
*9 7 2 5 4*
*Sorted list:*
*2 4 5 7 9*

### c. Program to remove duplicate elements from a singly linked list

Algorithm

1. Create a class Node which has two attributes: data and next. Next is a pointer to the next node in the list.
2. Create another class RemoveDuplicate which has two attributes: head and tail.
3. addNode() will add a new node to the list:
    1. Create a new node.
    2. It first checks, whether the head is equal to null which means the list is empty.
    3. If the list is empty, both head and tail will point to a newly added node.
    4. If the list is not empty, the new node will be added to end of the list such that tail's next will point to a newly added node. This new node will become the new tail of the list.
4. removeDuplicate() will remove duplicate nodes from the list.
    1. Define a new node current which will initially point to head.
    2. Node temp will point to current and index will always point to node next to current.
    3. Loop through the list till current points to null.
    4. Check whether current?s data is equal to index's data that means index is duplicate of current.
    5. Since index points to duplicate node so skip it by making node next to temp to will point to node next to index, i.e. temp.next = index.next.
5. display() will display the nodes present in the list:
    1. Define a node current which will initially point to the head of the list.
    2. Traverse through the list till current points to null.
    3. Display each node by making current to point to node next to it in each iteration.

---

```
1.  public class RemoveDuplicate {
2.
3.      //Represent a node of the singly linked list
4.      class Node{
5.          int data;
6.          Node next;
7.
8.          public Node(int data) {
9.              this.data = data;
10.             this.next = null;
11.         }
12.     }
13.
```

```
14.  //Represent the head and tail of the singly linked list
15.  public Node head = null;
16.  public Node tail = null;
17.
18.  //addNode() will add a new node to the list
19.  public void addNode(int data) {
20.     //Create a new node
21.     Node newNode = new Node(data);
22.
23.     //Checks if the list is empty
24.     if(head == null) {
25.        //If list is empty, both head and tail will point to new node
26.        head = newNode;
27.        tail = newNode;
28.     }
29.     else {
30.        //newNode will be added after tail such that tail's next will point to newNode
31.        tail.next = newNode;
32.        //newNode will become new tail of the list
33.        tail = newNode;
34.     }
35.  }
36.
37.  //removeDuplicate() will remove duplicate nodes from the list
38.  public void removeDuplicate() {
39.     //Node current will point to head
40.     Node current = head, index = null, temp = null;
41.
42.     if(head == null) {
43.        return;
44.     }
45.     else {
46.        while(current != null){
47.           //Node temp will point to previous node to index.
48.           temp = current;
49.           //Index will point to node next to current
50.           index = current.next;
51.
52.           while(index != null) {
53.              //If current node's data is equal to index node's data
54.              if(current.data == index.data) {
55.                 //Here, index node is pointing to the node which is duplicate of current node
56.                 //Skips the duplicate node by pointing to next node
57.                 temp.next = index.next;
58.              }
59.              else {
60.                 //Temp will point to previous node of index.
```

```
                temp = index;
            }
            index = index.next;
        }
        current = current.next;
    }
  }
}

//display() will display all the nodes present in the list
public void display() {
  //Node current will point to head
  Node current = head;
  if(head == null) {
    System.out.println("List is empty");
    return;
  }
  while(current != null) {
    //Prints each node by incrementing pointer
    System.out.print(current.data + " ");
    current = current.next;
  }
  System.out.println();
}

public static void main(String[] args) {

  RemoveDuplicate sList = new RemoveDuplicate();

  //Adds data to the list
  sList.addNode(1);
  sList.addNode(2);
  sList.addNode(3);
  sList.addNode(2);
  sList.addNode(2);
  sList.addNode(4);
  sList.addNode(1);

  System.out.println("Originals list: ");
        sList.display();

        //Removes duplicate nodes
        sList.removeDuplicate();

        System.out.println("List after removing duplicates: ");
        sList.display();
    }
```

```
108.        }
```

**Output:**

*Originals list:*
*1 2 3 2 2 4 1*
*List after removing duplicates:*
*1 2 3 4*

### d. Merge two linked lists

- Given two sorted linked lists consisting of **N** and **M** nodes respectively. The task is to merge both of the lists (in place) and return the head of the merged list.
- **Examples:**
- ***Input: a:*** *5->10->15,* ***b:*** *2->3->20*
  ***Output:*** *2->3->5->10->15->20*
- ***Input: a:*** *1->1,* ***b:*** *2->4*
  ***Output:*** *1->1->2->4*

  ***Input: a:*** *1->2->3,* ***b:*** *null*
  ***Output:*** *1->2->3*

**Brute Force Way:**
In this Approach we use an array of (n+m) size where n and m are length respective linked list and then store all the element in vector and then we sort the vector and make new linked list it will be our answer.

```
// Java program to merge two sorted linked list by storing
// them into array
import java.util.ArrayList;
import java.util.Collections;

class Node {
   int data;
   Node next;
   Node(int x) {
      this.data = x;
      this.next = null;
   }
}

class GfG {

   // Function to merge two sorted linked list
   static Node mergeSortedList(Node a, Node b) {

      // ArrayList to temporarily store the values
```

```
    ArrayList<Integer> list = new ArrayList<>();

    // Pushing the values of a linked list
    while (a != null) {
        list.add(a.data);
        a = a.next;
    }

    // Pushing the values of b linked list
    while (b != null) {
        list.add(b.data);
        b = b.next;
    }

    // Sorting the list
    Collections.sort(list);

    // Creating a new list with sorted values
    Node temp = new Node(-1);
    Node head = temp;
    for (int value : list) {
        temp.next = new Node(value);
        temp = temp.next;
    }
    head = head.next;

    // Returning the resultant linked list
    return head;
}

// Driver code
public static void main(String[] args) {

    // Create a hard-coded linked list:
    // 2 -> 4 -> 8 -> 9
    Node a = new Node(2);
    a.next = new Node(4);
    a.next.next = new Node(8);
    a.next.next.next = new Node(9);

    // Create another hard-coded linked list:
    // 1 -> 3 -> 8 -> 10
    Node b = new Node(1);
    b.next = new Node(3);
    b.next.next = new Node(8);
    b.next.next.next = new Node(10);
```

```
        Node res = mergeSortedList(a, b);

        Node temp = res;
          System.out.print("Merged Link List is:\n");
        while (temp != null) {
           System.out.print(temp.data + " ");
           temp = temp.next;
        }
      }
    }
```

**Output**

```
Merged Link List is:

1 2 3 4 8 8 9 10
```

**Time Complexity:O((n+m)+(n+m)*log(n+m)),**(n+m)for traversing linked lists and (n+m)*log(n+m) for sorting array which can be further solved to **O((n+m)*log(n+m)).**
**Auxiliary Space: O(n+m)**

**Program 3**

**write a java program to perform various operations on doubly linked list**

```
import java.util.*;
import java.lang.*;

// Java program to implement all functions
// used in Doubly Linked List

// Node for Doubly Linked List
class node{
	int key;
	node prev;
	node next;
	node(){
		prev = null;
		next = null;
	}
}

class Main{
	static node head = null;
	static node first = null;
	static node tail = null;
	static node temp = null;
```

```
static int i = 0;
// Function to add a node in the
// Doubly Linked List
static void addnode(int k)
{

        // Allocating memory
        // to the Node ptr
        node ptr = new node();

        // Assign Key to value k
        ptr.key = k;

        // Next and prev pointer to NULL
        ptr.next = null;
        ptr.prev = null;

        // If Linked List is empty
        if (head == null) {
                head = ptr;
                first = head;
                tail = head;
        }

        // Else insert at the end of the
        // Linked List
```

```
        else {
                temp = ptr;
                first.next = temp;
                temp.prev = first;
                first = temp;
                tail = temp;
        }

        // Increment for number of Nodes
        // in the Doubly Linked List
        i++;
}

// Function to traverse the Doubly
// Linked List
static void traverse()
{
        // Nodes points towards head node
        node ptr = head;

        // While pointer is not NULL,
        // traverse and print the node
        while (ptr != null) {

                // Print key of the node
                System.out.print( ptr.key+" ");
```

```
            ptr = ptr.next;
        }
        System.out.println();
}

// Function to insert a node at the
// beginning of the linked list
static void insertatbegin(int k)
{

        // Allocating memory
        // to the Node ptr
        node ptr = new node();

        // Assign Key to value k
        ptr.key = k;

        // Next and prev pointer to NULL
        ptr.next = null;
        ptr.prev = null;

        // If head is NULL
        if (head == null) {
            first = ptr;
            first = head;
            tail = head;
```

```
        }

        // Else insert at beginning and
        // change the head to current node
        else {
                temp = ptr;
                temp.next = head;
                head.prev = temp;
                head = temp;
        }
        i++;
}

// Function to insert Node at end
static void insertatend(int k)
{

        // Allocating memory
        // to the Node ptr
        node ptr= new node();

        // Assign Key to value k
        ptr.key = k;

        // Next and prev pointer to NULL
        ptr.next = null;
```

```
        ptr.prev = null;

        // If head is NULL
        if (head == null) {
                first = ptr;
                first = head;
                tail = head;
        }

        // Else insert at the end
        else {
                temp = ptr;
                temp.prev = tail;
                tail.next = temp;
                tail = temp;
        }
        i++;
}

// Function to insert Node at any
// position pos
static void insertatpos(int k, int pos)
{

        // For Invalid Position
        if (pos < 1 || pos > i + 1) {
```

```
            System.out.println("Please enter a valid position");
}

// If position is at the front,
// then call insertatbegin()
else if (pos == 1) {
            insertatbegin(k);
}

// Position is at length of Linked
// list + 1, then insert at the end
else if (pos == i + 1) {
            insertatend(k);
}

// Else traverse till position pos
// and insert the Node
else {
            node src = head;

            // Move head pointer to pos
            while (pos--!=0) {
                        src = src.next;
            }

            // Allocate memory to new Node
```

```
                node da, ba;
                node ptr = new node();
                ptr.next = null;
                ptr.prev = null;
                ptr.key = k;

                // Change the previous and next
                // pointer of the nodes inserted
                // with previous and next node
                ba = src;
                da = (src.prev);
                ptr.next = (ba);
                ptr.prev = (da);
                da.next = ptr;
                ba.prev = ptr;
                i++;
        }
}

// Function to delete node at the
// beginning of the list
static void delatbegin()
{
        // Move head to next and
        // decrease length by 1
        head = head.next;
```

```
        i--;
}

// Function to delete at the end
// of the list
static void delatend()
{
        // Mode tail to the prev and
        // decrease length by 1
        tail = tail.prev;
        tail.next = null;
        i--;
}

// Function to delete the node at
// a given position pos
static void delatpos(int pos)
{

        // If invalid position
        if (pos < 1 || pos > i + 1) {
                System.out.println("Please enter a valid position");
        }

        // If position is 1, then
        // call delatbegin()
```

```
else if (pos == 1) {
        delatbegin();
}

// If position is at the end, then
// call delatend()
else if (pos == i) {
        delatend();
}

// Else traverse till pos, and
// delete the node at pos
else {
        // Src node to find which
        // node to be deleted
        node src = head;
        pos--;

        // Traverse node till pos
        while (pos--!=0) {
                src = src.next;
        }

        // previous and after node
        // of the src node
        node pre, aft;
```

```
                pre = (src.prev);
                aft = (src.next);

                // Change the next and prev
                // pointer of pre and aft node
                pre.next = (aft);
                aft.prev = (pre);

                // Decrease the length of the
                // Linked List
                i--;
            }
        }
        public static void main(String args[])
        {

            // Adding node to the linked List
            addnode(2);
            addnode(4);
            addnode(9);
            addnode(1);
            addnode(21);
            addnode(22);

            // To print the linked List
            System.out.print("Linked List: ");
```

```
traverse();
System.out.println();

// To insert node at the beginning
insertatbegin(1);
System.out.print("Linked List after inserting 1 at beginning: ");
traverse();

// To insert at the end
insertatend(0);
System.out.print("Linked List after inserting 0 at end: ");
traverse();

// To insert Node with
// value 44 after 3rd Node
insertatpos(44, 3);
System.out.print("Linked List after inserting 44 after 3rd Node: ");
traverse();
System.out.println();

// To delete node at the beginning
delatbegin();
System.out.print("Linked List after deleting node at beginning: ");
traverse();

// To delete at the end
```

```
            delatend();
            System.out.print("Linked List after deleting node at end: ");
            traverse();

            // To delete Node at position 5
            System.out.print("Linked List after deleting node at position 5: ");
            delatpos(5);
            traverse();
        }
}
```

**Output**

```
Linked List: 2 4 9 1 21 22

Linked List after inserting 1 at beginning: 1 2 4 9 1 21 22
Linked List after inserting 0 at end: 1 2 4 9 1 21 22 0
Linked List after inserting 44 after 3rd Node: 1 2 4 44 9 1 21 22 0

Linked List after deleting node at beginning: 2 4 44 9 1 21 22 0
Linked List after deleting node at end: 2 4 44 9 1 21 22
Linked List after deleting node at position 5: 2 4 44 9 21 22
```

**Time and Space of Complexity of Each Operation :**
**Add a node at the front of DLL:**
**Time Complexity:** O(1)
**Space Complexity**: O(1)
**Traversal of a Doubly linked list:**
**Time Complexity:** O(n)
**Space Complexity:** O(1)
**Insertion of a node at the beginning:**
**Time Complexity:** O(1)
**Space Complexity:** O(1)
**Insertion of a node at the end:**
**Time Complexity:** O(1)
**Space Complexity:** O(1)

**Insertion of a node at a given position:**
**Time Complexity:** O(n)
**Space Complexity:** O(1)
**Deletion of a node at the beginning:**
**Time Complexity:** O(1)
**Space Complexity:** O(1)
**Deletion of a node at the end:**
**Time Complexity:** O(1)
**Space Complexity:** O(1)
**Deletion of a node given the position:**
**Time Complexity:** O(n)
**Space Complexity:** O(1)

**Program 4**

**Write a java program to perform various operations on circular linked list.**

**Steps to Implementation of Cirular LinkedList**

1. **Define the node class**: Create the class to represents the each element(node) in the circular Linked List and each node can contains the data and the reference to the next node.
2. **Define the CircularLinkedList class**: Create the class to manage the Circular Linked List and this class can contains the methods for the insertion, deletion, traversal and any other necessary operations.
3. **Implements the insertion methods**: We can add the methods to the insert the nodes at the beginging and end of the list. Ensure that next pointer of the last node always points back to the head to maintains the circular structure.
4. **Implements the deletion methods**: We can add the methods to delete the nodes from the begining and the end of list and it can update the head and tail pointers.
5. **Implements the traversal**: We can add the method to the traverse the list and display its the elements of the list.
6. Finally, We can write the main method to create the instance of CirularLinkedlist class and implements the performing the various operations such as the insertion, deletion and traversal of the Circular Linked List.

```
// Java Program for Implementing
// Circular Linked List

// Class Node
class Node {
   int data;
   Node next;

   // Constructor to initialize a node with data
   public Node(int data) {
      this.data = data;
      this.next = null;
   }
}

// Class CircularLinkedList
class CircularLinkedList {
   private Node head;
   private Node tail;

   // Constructor CircularLinkedList
   public CircularLinkedList() {
      head = null;
      tail = null;
   }
```

```
// Method to insert a new node with the given
  // data into the circular linked list
public void insert(int data) {
   Node newNode = new Node(data);

   // If the list is empty, make the new node
     // the head and tail
   if (head == null) {
      head = newNode;
      tail = newNode;

      // Point to itself in a circular list
      newNode.next = head;
   } else {
      newNode.next = head;
      tail.next = newNode;
      tail = newNode;
   }
}

// Method to delete the node with the given
  // key from the circular linked list
public void delete(int key) {
   if (head == null) {
      return;
   }

   Node curr = head;
   Node prev = null;
   while (curr.next != head) {
      if (curr.data == key) {
         // If the node to be deleted is the head node
         if (prev == null) {
            Node last = head;
            while (last.next != head) {
               last = last.next;
            }
            head = curr.next;
            last.next = head;
            return;
         } else {
            prev.next = curr.next;

            // Update tail if the last node is deleted
            if (curr == tail) {
               tail = prev;
```

```
                }
                return;
            }
        }
        prev = curr;
        curr = curr.next;
    }

    // Check if the node to be deleted is the last node
    if (curr == head && curr.data == key) {
        prev.next = head;

          // Update tail if the last node is deleted
        tail = prev;
    }
  }

  // Method to display the elements
   // of the circular linked list
  public void printList() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }

     System.out.print("Linked List Elements : ");

    Node itr = head;
    do {
        System.out.print(itr.data + " ");
        itr = itr.next;
    } while (itr != head);
    System.out.println();
  }
}

public class MyClass {
  public static void main(String[] args) {
    // Create a circular linked list
    CircularLinkedList list = new CircularLinkedList();

    // Insert elements into the list
    list.insert(1);
    list.insert(2);
    list.insert(3);
    list.insert(4);
```

```
        // Print the list
          // Output: 1 2 3 4
        list.printList();

        // Delete an element from the list
        list.delete(2);

        // Print the updated list
          // Output: 1 3 4
        list.printList();
    }
}
```

**Output**

```
Linked List Elements : 1 2 3 4

Linked List Elements : 1 3 4
```

**Complexity of the Above Method:**

**Time Complexity:**

- *The overall time complexity for the most operations like insertion at begining, insertion at end, deletion at begining and traversal is O(1) or O(n) where the n is number of the element in list.*
- *Insertion and deletion operations at the begining and end of list have the constant time complexity is O(1) as they can involves the fixed number of operations regardless of list size.*
- *Deletion at end and traversal operations have the linear time complexity is O(n) as they can require the traversing the entire list which grows with the number of the elements in list.*

**Space Complexity:**

- *Overall space complexity of the circular Linked List implementation is O(n) where n is the number of the elements in list.*
- *This space complexity can required to stores the node of linked list.*
- *The space complexity of the individual methods is constant O(1) as they can required only fixed amount of memory for the variables and temporary nodes of the list.*

**Program 5**

**Write a java program for performing various operation son stack using linked list.**

```
// Class representing a node in the linked list
class Node {
    int data;
    Node next;
    Node(int new_data) {
        this.data = new_data;
        this.next = null;
    }
}

// Class to implement stack using a singly linked list
class Stack {

    // Head of the linked list
    Node head;

    // Constructor to initialize the stack
    Stack() { this.head = null; }

    // Function to check if the stack is empty
    boolean isEmpty() {

        // If head is null, the stack is empty
```

```
    return head == null;
}

// Function to push an element onto the stack
void push(int new_data) {

    // Create a new node with given data
    Node new_node = new Node(new_data);

    // Check if memory allocation for the new node
    // failed
    if (new_node == null) {
        System.out.println("\nStack Overflow");
        return;
    }

    // Link the new node to the current top node
    new_node.next = head;

    // Update the top to the new node
    head = new_node;
}

// Function to remove the top element from the stack
void pop() {
```

```
    // Check for stack underflow
    if (isEmpty()) {
        System.out.println("\nStack Underflow");
        return;
    }
    else {

        // Assign the current top to a temporary
        // variable
        Node temp = head;

        // Update the top to the next node
        head = head.next;

        // Deallocate the memory of the old top node
        temp = null;
    }
}

// Function to return the top element of the stack
int peek() {

    // If stack is not empty, return the top element
    if (!isEmpty())
        return head.data;
    else {
```

```
            System.out.println("\nStack is empty");
            return Integer.MIN_VALUE;
        }
    }
}

// Driver code
public class Main {
    public static void main(String[] args)
    {
        // Creating a stack
        Stack st = new Stack();

        // Push elements onto the stack
        st.push(11);
        st.push(22);
        st.push(33);
        st.push(44);

        // Print top element of the stack
        System.out.println("Top element is " + st.peek());

        // removing two elemements from the top
          System.out.println("Removing two elements...");
        st.pop();
        st.pop();
```

```
        // Print top element of the stack

        System.out.println("Top element is " + st.peek());

    }

}
```

**Output**

```
Top element is 44

Top element is 22
```

**Program 6**

**Queue – Linked List Implementation**

```
// Java program to implement the queue data structure using
// linked list

// Node class representing a single node in the linked list
class Node {
   int data;
   Node next;

   Node(int new_data) {
      this.data = new_data;
      this.next = null;
   }
}

// Class to implement queue operations using a linked list
class Queue {

   // Pointer to the front and the rear of the linked list
   Node front, rear;

   // Constructor to initialize the front and rear
   Queue() { front = rear = null; }
```

```
// Function to check if the queue is empty
boolean isEmpty() {

    // If the front and rear are null, then the queue is
    // empty, otherwise it's not
    return front == null && rear == null;
}

// Function to add an element to the queue
void enqueue(int new_data) {

    // Create a new linked list node
    Node new_node = new Node(new_data);

    // If queue is empty, the new node is both the front
    // and rear
    if (rear == null) {
        front = rear = new_node;
        return;
    }

    // Add the new node at the end of the queue and
    // change rear
    rear.next = new_node;
    rear = new_node;
}
```

```
// Function to remove an element from the queue
void dequeue() {

    // If queue is empty, return
    if (isEmpty()) {
        System.out.println("Queue Underflow");
        return;
    }

    // Store previous front and move front one node
    // ahead
    Node temp = front;
    front = front.next;

    // If front becomes null, then change rear also
    // to null
    if (front == null) {
        rear = null;
    }
}

// Function to get the front element of the queue
int getFront() {

    // Checking if the queue is empty
```

```
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return Integer.MIN_VALUE;
        }
        return front.data;
    }

    // Function to get the rear element of the queue
    int getRear() {

        // Checking if the queue is empty
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return Integer.MIN_VALUE;
        }
        return rear.data;
    }
}

// Driver code to test the queue implementation
public class Main {
    public static void main(String[] args) {
        Queue q = new Queue();

        // Enqueue elements into the queue
        q.enqueue(10);
```

```
        q.enqueue(20);

         System.out.println("Queue Front: " + q.getFront());
        System.out.println("Queue Rear: " + q.getRear());

        // Dequeue elements from the queue
        q.dequeue();
        q.dequeue();

        // Enqueue more elements into the queue
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);

        // Dequeue an element from the queue
        q.dequeue();

        System.out.println("Queue Front: " + q.getFront());
        System.out.println("Queue Rear: " + q.getRear());
    }
}
```

**Output**

```
Queue Front: 10
Queue Rear: 20
Queue Front: 40
Queue Rear: 50
```

**Program 7**

**Write a java program for the following using stack**

**a) Infix to postfix conversion.**

**b) Expression evaluation.**

**c) Obtain the binary number for a given decimal number.**

```
import java.util.Stack;


class GfG {


    // Function to return precedence of operators
  int prec(char c) {
      if (c == '^')
          return 3;
      else if (c == '/' || c == '*')
          return 2;
      else if (c == '+' || c == '-')
          return 1;
      else
          return -1;
  }


    // Function to perform infix to postfix conversion
  void infixToPostfix(String s) {
      Stack<Character> st = new Stack<>();
      StringBuilder result = new StringBuilder();
```

```
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);

    // If the scanned character is
    // an operand, add it to the output string.
    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
        result.append(c);

    // If the scanned character is
    // an ‘(‘, push it to the stack.
    else if (c == '(')
        st.push('(');

    // If the scanned character is an ‘)’,
    // pop and add to the output string from the stack
    // until an ‘(‘ is encountered.
    else if (c == ')') {
        while (st.peek() != '(') {
            result.append(st.pop());
        }
        st.pop();
    }

    // If an operator is scanned
    else {
        while (!st.isEmpty() && (prec(c) < prec(st.peek()) ||
```

```
                    prec(c) == prec(st.peek()))) {
                result.append(st.pop());
            }
            st.push(c);
        }
    }

    // Pop all the remaining elements from the stack
    while (!st.isEmpty()) {
        result.append(st.pop());
    }

    System.out.println(result.toString());
  }

  public static void main(String[] args) {
    GfG gfg = new GfG();
    String exp = "a+b*(c^d-e)^(f+g*h)-i";
    gfg.infixToPostfix(exp);
  }
}
```

**Output**

```
abcd^e-fgh*+^*+i-
```

**Time Complexity:** O(N), where N is the size of the infix expression
**Auxiliary Space:** O(N), where N is the size of the infix expression

**Program 8**

**Java Program for Binary Search (Recursive and Iterative)**

```
// Java Program to Illustrate
// Iterative Binary Search

// Main class
// BinarySearch
class GFG {

    // Method 1
    // Returns index of x
    // if it is present in arr[],
    // else return -1
    int binarySearch(int arr[], int x)
    {

        int l = 0, r = arr.length - 1;

        // Checking element in whole array
        while (l <= r) {
            int m = l + (r - l) / 2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;
```

```
                // If x greater, ignore left half
                if (arr[m] < x)
                        l = m + 1;

                // If x is smaller,
                // element is on left side
                // so ignore right half
                else
                        r = m - 1;
        }

        // If we reach here,
        // element is not present
        return -1;
}

// Method 2
// Main driver method
public static void main(String args[])
{

        GFG ob = new GFG();

        // Input array
        int arr[] = { 2, 3, 4, 10, 40 };
```

```
        // Length of array
        int n = arr.length;
        // Element to be checked if present or not
        int x = 10;

        // Calling the method 1 and
        // storing result
        int result = ob.binarySearch(arr, x);

        // Element present
        if (result == -1)

            // Print statement
            System.out.println("Element not present");

        // Element not present
        else

            // Print statement
            System.out.println("Element found at index "
                                        + result);
    }
}
```

**Output**

```
Element found at index 3
```

**Time Complexity**: O(log n)
**Auxiliary Space**: O(1)

**Experiment 9: Write a java program to implement the following for a graph. a) BFS b) DFS**

**a. Java Program for bfs for a Graph**

```
import java.util.*;

class GfG {

    // BFS from given source s
    static void bfs(List<List<Integer>> adj, int s) {

        // Create a queue for BFS
        Queue<Integer> q = new LinkedList<>();

        // Initially mark all the vertices as not visited
        // When we push a vertex into the q, we mark it as
        // visited
        boolean[] visited = new boolean[adj.size()];

        // Mark the source node as visited and enqueue it
        visited[s] = true;
        q.add(s);

        // Iterate over the queue
        while (!q.isEmpty()) {
```

```
        // Dequeue a vertex and print it
        int curr = q.poll();
        System.out.print(curr + " ");

        // Get all adjacent vertices of the dequeued vertex
        // If an adjacent has not been visited, mark it
        // visited and enqueue it
        for (int x : adj.get(curr)) {
            if (!visited[x]) {
                visited[x] = true;
                q.add(x);
            }
        }
    }
}

// Function to add an edge to the graph
static void addEdge(List<List<Integer>> adj, int u, int v) {
    adj.get(u).add(v);
    adj.get(v).add(u); // Undirected graph
}

public static void main(String[] args) {

    // Number of vertices in the graph
    int V = 5;
```

```
        // Adjacency list representation of the graph
        List<List<Integer>> adj = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }

        // Add edges to the graph
        addEdge(adj, 0, 1);
        addEdge(adj, 0, 2);
        addEdge(adj, 1, 3);
        addEdge(adj, 1, 4);
        addEdge(adj, 2, 4);

        // Perform BFS traversal starting from vertex 0
        System.out.println("BFS starting from 0:");
        bfs(adj, 0);
    }
}
```

**Output**

```
BFS starting from 0 :
0 1 2 3 4
```

**Time Complexity:** O(V+E), where V is the number of nodes and E is the number of edges.
**Auxiliary Space:** O(V)

**b.**

**Java Program for Depth First Search for a Graph**

```
// Java program to print DFS traversal
// from a given graph
import java.io.*;
import java.util.*;

// This class represents a
// directed graph using adjacency
// list representation
class Graph {
   private int V;

   // Array of lists for
   // Adjacency List Representation
   private LinkedList<Integer> adj[];

   // Constructor
   @SuppressWarnings("unchecked") Graph(int v)
   {
      V = v;
      adj = new LinkedList[v];
      for (int i = 0; i < v; ++i)
         adj[i] = new LinkedList();
   }
```

```
// Function to add an edge into the graph
void addEdge(int v, int w)
{
   // Add w to v's list.
   adj[v].add(w);
}

// A function used by DFS
void DFSUtil(int v, boolean visited[])
{
   // Mark the current node as visited and print it
   visited[v] = true;
   System.out.print(v + " ");

   // Recur for all the vertices adjacent to this
   // vertex
   Iterator<Integer> i = adj[v].listIterator();
   while (i.hasNext()) {
      int n = i.next();
      if (!visited[n])
         DFSUtil(n, visited);
   }
}

// The function to do DFS traversal.
```

```
// It uses recursive DFSUtil()
void DFS(int v)
{
    // Mark all the vertices as
    // not visited(set as
    // false by default in java)
    boolean visited[] = new boolean[V];

    // Call the recursive helper
    // function to print DFS
    // traversal
    DFSUtil(v, visited);
}

// Driver Code
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
```

```
        System.out.println(
            "Following is Depth First Traversal "
            + "(starting from vertex 2)");


        // Function call
        g.DFS(2);
    }
}
```

**Output**

```
Following is Depth First Traversal (starting from vertex 2):
2 0 1 3
```

***Complexity of the program above:***

***Time Complexity:*** *O(V+E) where V is the number of vertices in the graph and E is the number of edges.*

***Auxiliary Space:*** *O(V+E)*

**Program 10**

**Write a java program to implement Merge & Heap Sort of given elements.**

```
import java.util.*;

public class HeapSortUsingSTL {

	// Function to perform the heap sort
	public static void heapSort(int[] arr)
	{
		PriorityQueue<Integer> maxHeap
			= new PriorityQueue<>(
				Collections.reverseOrder());
		for (int i = 0; i < arr.length; i++) {
			maxHeap.offer(arr[i]);
		}
		for (int i = arr.length - 1; i >= 0; i--) {
			arr[i] = maxHeap.poll();
		}
	}

	// Driver Code
	public static void main(String[] args)
	{
		int[] arr = { 60, 20, 40, 70, 30, 10 };
		System.out.println("Before Sorting: "
```

```
                                                + Arrays.toString(arr));
                heapSort(arr);
                System.out.println("After Sorting: "
                                                + Arrays.toString(arr));
        }
}
```

**Output**

```
Before Sorting: [60, 20, 40, 70, 30, 10]
After Sorting: [10, 20, 30, 40, 60, 70]
```

***Complexity of the above program:***
***Time Complexity:*** *O(n log n)*
***Auxiliary Space:*** *O(n)*

**Program of Merge Sort in Java**

```
// Java program for Merge Sort

class MergeSort {

   // Merges two subarrays of a[]
   void merge(int a[], int l, int m, int r)
   {

       int n1 = m - l + 1;
      int n2 = r - m;
```

```
int L[] = new int[n1];

int R[] = new int[n2];


for (int i = 0; i < n1; ++i)

    L[i] = a[l + i];


  for (int j = 0; j < n2; ++j)

    R[j] = a[m + 1 + j];


// Merge the temp arrays

// Initial indexes of first and second subarrays

int i = 0, j = 0;


int k = l;

while (i < n1 && j < n2) {

    if (L[i] <= R[j]) {

        a[k] = L[i];

        i++;

    }

    else {

        a[k] = R[j];

        j++;

    }

    k++;

}
```

```
    while (i < n1) {
        a[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        a[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts a[l..r] using
// merge()
void sort(int a[], int l, int r)
{
    if (l < r) {

        int m = (l + r) / 2;

        // Sort first and second halves
        sort(a, l, m);
        sort(a, m + 1, r);

        // Merge the sorted halves
```

```
            merge(a, l, m, r);
        }
    }

    // Driver method
    public static void main(String args[])
    {
        int a[] = { 12, 11, 13, 5, 6, 7 };

        // Calling of Merge Sort
        MergeSort ob = new MergeSort();
        ob.sort(a, 0, a.length - 1);

        int n = a.length;
        for (int i = 0; i < n; ++i)
            System.out.print(a[i] + " ");
    }
}
```

**Output**

```
5 6 7 11 12 13
```

***The complexity of the above method:***

***Time Complexity****: O(n log n)*

***Auxiliary Space:*** *O(n)*

**Program 11**

**Write a java program to implement Quick Sort of given elements.**

```
import java.util.Arrays;

class Quicksort {

  // method to find the partition position
  static int partition(int array[], int low, int high) {

    // choose the rightmost element as pivot
    int pivot = array[high];

    // pointer for greater element
    int i = (low - 1);

    // traverse through all elements
    // compare each element with pivot
    for (int j = low; j < high; j++) {
      if (array[j] <= pivot) {

        // if element smaller than pivot is found
        // swap it with the greater element pointed by i
        i++;

        // swapping element at i with element at j
```

```
      int temp = array[i];
      array[i] = array[j];
      array[j] = temp;
    }

  }

  // swap the pivot element with the greater element specified by i
  int temp = array[i + 1];
  array[i + 1] = array[high];
  array[high] = temp;

  // return the position from where partition is done
  return (i + 1);
}

static void quickSort(int array[], int low, int high) {
  if (low < high) {

    // find pivot element such that
    // elements smaller than pivot are on the left
    // elements greater than pivot are on the right
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);
```

```
    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
 }

 public static void main(String args[]) {

  int[] data = { 8, 7, 2, 1, 0, 9, 6 };
  System.out.println("Unsorted Array");
  System.out.println(Arrays.toString(data));

  int size = data.length;

  // call quicksort() on array data
  quickSort(data, 0, size - 1);

  System.out.println("Sorted Array in Ascending Order ");
  System.out.println(Arrays.toString(data));
 }
}
```

**Output**

```
Unsorted Array
[8, 7, 2, 1, 0, 9, 6]
Sorted Array in Ascending Order
[0, 1, 2, 6, 7, 8, 9]
```

**Program 12**

**Write a java program to implement various operations on AVL trees**

```
// Java Program to Implement AVL Tree
class Node {
   int key, height;
   Node left, right;

   Node(int d) {
      key = d;
      height = 1;
   }
}

class AVLTree {

   Node root;

   // Get the height of the node
   int height(Node N) {
      if (N == null)
         return 0;
      return N.height;
   }

   // Get maximum of two integers
   int max(int a, int b) {
```

```
    return (a > b) ? a : b;
}

// Right rotate subtree rooted with y
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(height(x.left), height(x.right)) + 1;

    // Return new root
    return x;
}

// Left rotate subtree rooted with x
Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
```

```
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = max(height(x.left), height(x.right)) + 1;
    y.height = max(height(y.left), height(y.right)) + 1;

    // Return new root
    return y;
}

// Get balance factor of node N
int getBalance(Node N) {
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}

// Insert a key into the subtree rooted with node
        // and returns the new root of the subtree
Node insert(Node node, int key) {
    // Perform the normal BST insertion
    if (node == null)
        return (new Node(key));

    if (key < node.key)
```

```
    node.left = insert(node.left, key);
else if (key > node.key)
    node.right = insert(node.right, key);
else // Duplicate keys not allowed
    return node;

// Update height of this ancestor node
node.height = 1 + max(height(node.left), height(node.right));

// Get the balance factor of this ancestor node
     // to check whether this node became unbalanced
int balance = getBalance(node);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && key < node.left.key)
    return rightRotate(node);

// Right Right Case
if (balance < -1 && key > node.right.key)
    return leftRotate(node);

// Left Right Case
if (balance > 1 && key > node.left.key) {
    node.left = leftRotate(node.left);
```

```
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    // Return the (unchanged) node pointer
    return node;
}

// Utility function to print preorder traversal of the tree
void preOrder(Node node) {
    if (node != null) {
        System.out.print(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

// Utility function to print inorder traversal of the tree
void inOrder(Node node) {
    if (node != null) {
        inOrder(node.left);
```

```
        System.out.print(node.key + " ");
        inOrder(node.right);
    }
}

// Utility function to print postorder traversal of the tree
void postOrder(Node node) {
    if (node != null) {
        postOrder(node.left);
        postOrder(node.right);
        System.out.print(node.key + " ");
    }
}

public static void main(String[] args) {
    AVLTree tree = new AVLTree();

    // Insert nodes
    tree.root = tree.insert(tree.root, 10);
    tree.root = tree.insert(tree.root, 20);
    tree.root = tree.insert(tree.root, 30);
    tree.root = tree.insert(tree.root, 40);
    tree.root = tree.insert(tree.root, 50);
    tree.root = tree.insert(tree.root, 25);

    // Print preorder traversal of the tree
```

```
        System.out.println("Preorder traversal of constructed AVL tree is : ");
        tree.preOrder(tree.root);
        System.out.println();

        // Print inorder traversal of the tree
        System.out.println("Inorder traversal of constructed AVL tree is : ");
        tree.inOrder(tree.root);
        System.out.println();

        // Print postorder traversal of the tree
        System.out.println("Postorder traversal of constructed AVL tree is : ");
        tree.postOrder(tree.root);
        System.out.println();
    }
}
```

**Output:**

```
Preorder traversal of constructed AVL tree is :
30 20 10 25 40 50
Inorder traversal of constructed AVL tree is :
10 20 25 30 40 50
Postorder traversal of constructed AVL tree is :
10 25 20 50 40 30
```

**Program 13**

**Write a java program to perform the following operations: a) Insertion into a B-tree b) Searching in a B-tree**

```
class BTree {
    private static final int T = 3; // Minimum degree (defines the range for number of keys)

    // Node class to represent a B-tree node
    static class Node {
        int[] keys = new int[2 * T - 1];
        Node[] children = new Node[2 * T];
        int numKeys = 0;
        boolean isLeaf = true;

        // Constructor for a node
        Node() {
            for (int i = 0; i < 2 * T - 1; i++) {
                keys[i] = -1; // Initialize all keys as -1 (empty)
            }
            for (int i = 0; i < 2 * T; i++) {
                children[i] = null; // Initialize children as null
            }
        }
    }

    private Node root;
```

```
public BTree() {
    root = new Node();
}

// Method to insert a key into the B-tree
public void insert(int key) {
    Node rootNode = root;

    if (rootNode.numKeys == 2 * T - 1) {
        Node newNode = new Node();
        newNode.children[0] = rootNode;
        root = newNode;
        splitChild(newNode, 0);
        insertNonFull(newNode, key);
    } else {
        insertNonFull(rootNode, key);
    }
}

// Insert a key into a non-full node
private void insertNonFull(Node node, int key) {
    int i = node.numKeys - 1;

    if (node.isLeaf) {
        while (i >= 0 && key < node.keys[i]) {
```

```
            node.keys[i + 1] = node.keys[i];
            i--;
        }
        node.keys[i + 1] = key;
        node.numKeys++;
    } else {
        while (i >= 0 && key < node.keys[i]) {
            i--;
        }
        i++;

        if (node.children[i].numKeys == 2 * T - 1) {
            splitChild(node, i);
            if (key > node.keys[i]) {
                i++;
            }
        }
        insertNonFull(node.children[i], key);
    }
}

// Split the child of a node at index i
private void splitChild(Node parent, int i) {
    Node child = parent.children[i];
    Node newChild = new Node();
    newChild.isLeaf = child.isLeaf;
```

```
newChild.numKeys = T - 1;

for (int j = 0; j < T - 1; j++) {
   newChild.keys[j] = child.keys[j + T];
}

if (!child.isLeaf) {
   for (int j = 0; j < T; j++) {
      newChild.children[j] = child.children[j + T];
   }
}

child.numKeys = T - 1;

for (int j = parent.numKeys; j > i; j--) {
   parent.children[j + 1] = parent.children[j];
}

parent.children[i + 1] = newChild;

for (int j = parent.numKeys - 1; j >= i; j--) {
   parent.keys[j + 1] = parent.keys[j];
}

parent.keys[i] = child.keys[T - 1];
parent.numKeys++;
```

```
}

// Method to search for a key in the B-tree
public boolean search(int key) {
    return search(root, key);
}

// Search for a key in a specific node
private boolean search(Node node, int key) {
    int i = 0;
    while (i < node.numKeys && key > node.keys[i]) {
        i++;
    }

    if (i < node.numKeys && node.keys[i] == key) {
        return true; // Found the key
    }

    if (node.isLeaf) {
        return false; // Key not found in leaf node
    }

    return search(node.children[i], key); // Continue search in the child node
}

// Method to print the B-tree (for visualization)
```

```
public void print() {
    print(root, 0);
}

private void print(Node node, int level) {
    if (node != null) {
        System.out.println("Level " + level + " : ");
        for (int i = 0; i < node.numKeys; i++) {
            System.out.print(node.keys[i] + " ");
        }
        System.out.println();

        if (!node.isLeaf) {
            for (int i = 0; i <= node.numKeys; i++) {
                print(node.children[i], level + 1);
            }
        }
    }
}

// Main method to test B-tree operations
public static void main(String[] args) {
    BTree bTree = new BTree();

    System.out.println("Inserting elements into the B-tree:");
    int[] elements = {10, 20, 5, 6, 12, 30, 7, 17};
```

```
        for (int elem : elements) {
            bTree.insert(elem);
            bTree.print(); // Print the B-tree after each insertion
        }

        System.out.println("Searching for 12 in the B-tree: " + bTree.search(12));
        System.out.println("Searching for 25 in the B-tree: " + bTree.search(25));
    }
}
```

Output

```
Inserting elements into the B-tree:
Level 0 :
10
Level 1 :
5 10
Level 0 :
5 10 20
Level 0 :
5 6 10 20
Level 1 :
5 6 10 12 20
Level 0 :
5 6 10 12 17 20
Level 0 :
5 6 10 12 17 20 30
```

Level 1 :

5 6 7 10 12 17 20 30

Searching for 12 in the B-tree: true

Searching for 25 in the B-tree: false

**Experiment 14: Write a java program to implementation of recursive and non-recursive functions to Binary tree Traversals**

```
// Java program to print inorder traversal
// using stack.
import java.util.ArrayList;
import java.util.Stack;

class Node {
    int data;
    Node left, right;

    Node(int x) {
        data = x;
        left = null;
        right = null;
    }
}
class GfG {
    // Iterative function for inorder tree traversal
    static ArrayList<Integer> inOrder(Node root) {
        ArrayList<Integer> ans = new ArrayList<>();
        Stack<Node> s = new Stack<>();
        Node curr = root;

        while (curr != null || !s.isEmpty()) {
```

```
        // Reach the left most Node of the curr Node
        while (curr != null) {

            // Place pointer to a tree node on
            // the stack before traversing
            // the node's left subtree
            s.push(curr);
            curr = curr.left;
        }

        // Current must be NULL at this point
        curr = s.pop();
        ans.add(curr.data);

        // we have visited the node and its
        // left subtree. Now, it's right
        // subtree's turn
        curr = curr.right;
    }

    return ans;
}

static void printList(ArrayList<Integer> v) {
    for (int i : v) {
```

```
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {

        // Constructed binary tree is
        //         1
        //       /   \
        //     2      3
        //    / \
        //  4    5
        Node root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(4);
        root.left.right = new Node(5);

        ArrayList<Integer> res = inOrder(root);
        printList(res);
    }
}
```

**Output**

```
4 2 5 1 3
```

**Experiment 15: Write a java program to implement all the functions of Dictionary (ADT) using Hashing.**

```
import java.util.LinkedList;


class HashTable {
    private static final int TABLE_SIZE = 10; // Hash table size
    private LinkedList<KeyValuePair>[] table;


    // Constructor
    public HashTable() {
        table = new LinkedList[TABLE_SIZE];
        for (int i = 0; i < TABLE_SIZE; i++) {
            table[i] = new LinkedList<KeyValuePair>();
        }
    }


    // Inner class to represent a key-value pair
    static class KeyValuePair {
        String key;
        String value;


        KeyValuePair(String key, String value) {
            this.key = key;
            this.value = value;
        }
```

```
}

// Hash function to calculate index for a given key
private int hash(String key) {
    return key.hashCode() % TABLE_SIZE;
}

// Insert a key-value pair
public void insert(String key, String value) {
    int index = hash(key);
    LinkedList<KeyValuePair> bucket = table[index];

    // Check if the key already exists and update the value if so
    for (KeyValuePair pair : bucket) {
        if (pair.key.equals(key)) {
            pair.value = value;
            return;
        }
    }

    // If key does not exist, insert a new key-value pair
    bucket.add(new KeyValuePair(key, value));
}

// Search for a key in the hash table
public String search(String key) {
```

```
    int index = hash(key);
    LinkedList<KeyValuePair> bucket = table[index];

    for (KeyValuePair pair : bucket) {
        if (pair.key.equals(key)) {
            return pair.value;
        }
    }

    return null; // Return null if key is not found
}

// Delete a key-value pair
public void delete(String key) {
    int index = hash(key);
    LinkedList<KeyValuePair> bucket = table[index];

    KeyValuePair toRemove = null;
    for (KeyValuePair pair : bucket) {
        if (pair.key.equals(key)) {
            toRemove = pair;
            break;
        }
    }

    if (toRemove != null) {
```

```
            bucket.remove(toRemove);
        }
    }

    // Update the value for a key
    public void update(String key, String newValue) {
        int index = hash(key);
        LinkedList<KeyValuePair> bucket = table[index];

        for (KeyValuePair pair : bucket) {
            if (pair.key.equals(key)) {
                pair.value = newValue;
                return;
            }
        }

        System.out.println("Key not found, unable to update.");
    }

    // Print the entire hash table
    public void printTable() {
        for (int i = 0; i < TABLE_SIZE; i++) {
            if (table[i].isEmpty()) {
                continue;
            }
            System.out.print("Bucket " + i + ": ");
```

```
        for (KeyValuePair pair : table[i]) {
            System.out.print("[" + pair.key + " = " + pair.value + "] ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    HashTable dictionary = new HashTable();

    // Insert key-value pairs
    dictionary.insert("apple", "a fruit");
    dictionary.insert("dog", "a domestic animal");
    dictionary.insert("java", "a programming language");

    // Search for keys
    System.out.println("Search 'apple': " + dictionary.search("apple"));
    System.out.println("Search 'java': " + dictionary.search("java"));
    System.out.println("Search 'dog': " + dictionary.search("dog"));
    System.out.println("Search 'cat': " + dictionary.search("cat"));

    // Update value
    dictionary.update("dog", "a friendly domestic animal");
    System.out.println("Updated value for 'dog': " + dictionary.search("dog"));

    // Delete key
```

```
        dictionary.delete("apple");
        System.out.println("After deletion, search for 'apple': " + dictionary.search("apple"));

        // Print the current state of the hash table
        dictionary.printTable();
    }
}
```

Output

```
Search 'apple': a fruit
Search 'java': a programming language
Search 'dog': a domestic animal
Search 'cat': null
Updated value for 'dog': a friendly domestic animal
After deletion, search for 'apple': null
Bucket 1: [dog = a friendly domestic animal]
Bucket 4: [java = a programming language]
```